Module 14) Python – Collections, functions and Modules

Accessing List

Theory:

1) Understanding how to create and access elements in a list.

Answer ->In Python, a list is a collection of items that can store multiple values in a single variable. Lists are ordered, mutable (can be changed), and can contain elements of different data types such as integers, strings, or even other lists.

Example:

```
fruits = ["apple", "banana", "cherry"]

numbers = [1, 2, 3, 4, 5]

mixed = [10, "hello", 3.5, True]
```

2. Accessing Elements in a List

Each element in a list has an index number that starts from 0 for the first element.

Example:

```
fruits = ["apple", "banana", "cherry"]

print(fruits[0])  # Output: apple

print(fruits[1])  # Output: banana
```

- fruits[0] accesses the first element.
- fruits[1] accesses the second element.

3. Negative Indexing

Python also supports negative indexing, which allows access from the end of the list.

Example:

```
fruits = ["apple", "banana", "cherry"]

print(fruits[-1])  # Output: cherry

print(fruits[-2])  # Output: banana
```

4. Accessing a Range of Elements (Slicing)

You can access a subset of elements using slicing [start:end].

Example:

```
numbers = [10, 20, 30, 40, 50]

print(numbers[1:4])  # Output: [20, 30, 40]
```

- The slice starts from index 1 and ends before index 4.

5. Modifying Elements

Lists are mutable, meaning you can change values after creating them.

Example:

```
fruits = ["apple", "banana", "cherry"]

fruits[1] = "orange"

print(fruits)  # Output: ['apple', 'orange', 'cherry']
```

2) Indexing in lists (positive and negative indexing).

In Python, indexing means accessing elements of a list using their position (index number). Each element in a list has a unique index value that identifies its position.

1. Positive Indexing

- In positive indexing, counting starts from 0 for the first element, 1 for the second, and so on.
- It is used to access elements from the beginning of the list.

Example**:**

```
fruits = ["apple", "banana", "cherry", "mango"]

print(fruits[0])  # Output: apple

print(fruits[2])  # Output: cherry
```

Here,

- fruits[0] → first element
- fruits[1] → second element

2. Negative Indexing

- In negative indexing, counting starts from -1 for the last element, -2 for the second last, and so on.
- It is used to access elements from the end of the list.

Example:

```
fruits = ["apple", "banana", "cherry", "mango"]

print(fruits[-1])  # Output: mango

print(fruits[-3])  # Output: banana
```

Here,

- fruits[-1] → last element
- fruits[-2] → second last element

3) Slicing a list: accessing a range of elements.

Slicing in Python means accessing a range of elements from a list by specifying the start and end index values. It allows us to extract a portion of a list instead of accessing single elements.

The general syntax is:

```
list_name[start:end]
```

- start → index where the slice begins (included)
- end → index where the slice ends (excluded)

Example:

```
numbers = [10, 20, 30, 40, 50, 60]

print(numbers[1:4])   # Output: [20, 30, 40]
```

Here, the slice starts from index 1 (second element) and ends before index 4.

You can also skip the start or end index:

```
print(numbers[:3])   # Output: [10, 20, 30]   → from beginning to index 2

print(numbers[2:])   # Output: [30, 40, 50, 60] → from index 2 to end
```

You can also include a step value using the format [start:end:step]:

```
print(numbers[0:6:2])  # Output: [10, 30, 50]
```

Here, the step value 2 means every second element is taken.

2. List Operations

Theory:

1) Common list operations: concatenation, repetition, membership

Python provides several common operations that can be performed on lists, such as concatenation, repetition, and membership. These operations make it easier to work with multiple lists and their elements.

1. Concatenation (+ Operator)

- Concatenation means joining two or more lists together to form a new list.
- The + operator is used for this purpose.

Example:

```
list1 = [1, 2, 3]

list2 = [4, 5]

result = list1 + list2

print(result)   # Output: [1, 2, 3, 4, 5]
```

2. Repetition (* Operator)

- Repetition is used to repeat the elements of a list multiple times.
- The * operator is used for this.

Example:

```
list1 = [10, 20]

print(list1 * 3)   # Output: [10, 20, 10, 20, 10, 20]
```

3. Membership (in / not in Operator)

- The membership operators are used to check whether an element is present in a list or not.

Example:

```
fruits = ["apple", "banana", "mango"]

print("apple" in fruits)     # Output: True

print("grape" not in fruits) # Output: True
```

2) Understanding list methods like append(), insert(), remove(), pop().

thon provides several built-in list methods to add, modify, and delete elements easily. The most commonly used methods are append(), insert(), remove(), and pop().

1. append()

- The append() method adds an element to the end of the list.

Example:

```
fruits = ["apple", "banana"]

fruits.append("cherry")

print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

2. insert()

- The insert() method adds an element at a specific position (index) in the list.

Example:

```
fruits = ["apple", "cherry"]

fruits.insert(1, "banana")

print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

3. remove()

- The remove() method deletes the first matching element by its value.

Example:

```
fruits = ["apple", "banana", "cherry"]

fruits.remove("banana")

print(fruits)  # Output: ['apple', 'cherry']
```

4. pop()

- The pop() method removes an element by index and returns it.
- If no index is given, it removes the last element by default.

Example:

```
fruits = ["apple", "banana", "cherry"]

fruits.pop(1)
```

```
print(fruits)  # Output: ['apple', 'cherry']
```

3. Working with Lists

Theroy

1 Iterating over a list using loops

Iteration means accessing each element of a list one by one using a loop. In Python, we commonly use the for loop and the while loop to iterate over lists. This allows us to perform operations like displaying, modifying, or processing each element.

1. Using a for loop

The for loop is the most common and simple way to iterate through a list.

Example:

```
fruits = ["apple", "banana", "cherry"]

for item in fruits:

   print(item)
```

Output:

```
apple

banana

cherry
```

Here, the loop runs once for each element in the list and prints it.

2. Using a while loop

We can also use a while loop with an index to access list elements.

Example:

```
fruits = ["apple", "banana", "cherry"]

i = 0

while i < len(fruits):

   print(fruits[i])

   i += 1
```

This loop continues until all elements are printed using their index positions.

2) Sorting and reversing a list using sort(), sorted(), and reverse().

In Python, lists can be sorted and reversed easily using built-in methods like sort(), sorted(), and reverse(). These methods help organize list elements in a specific order.

1. sort()

- The sort() method sorts the elements of a list in ascending order by default.
- It changes (modifies) the original list.

Example:

```
numbers = [5, 2, 8, 1]

numbers.sort()

print(numbers)  # Output: [1, 2, 5, 8]
```

- To sort in descending order, use:

```
numbers.sort(reverse=True)

print(numbers)  # Output: [8, 5, 2, 1]
```

2. sorted()

- The sorted() function returns a new sorted list without changing the original one. Example:

```
numbers = [5, 2, 8, 1]

new_list = sorted(numbers)

print(new_list)   # Output: [1, 2, 5, 8]

print(numbers)    # Output: [5, 2, 8, 1]
```

3. reverse()

- The reverse() method simply reverses the order of the elements in the list.
- It does not sort the list, only flips the existing order.

Example:

```
numbers = [10, 20, 30, 40]

numbers.reverse()

print(numbers)  # Output: [40, 30, 20, 10]
```

3) Basic list manipulations: addition, deletion, updating, and slicing.

Lists in Python are mutable, meaning their elements can be added, deleted, updated, or sliced easily. These basic operations allow us to modify and manage list data efficiently.

1. Addition of Elements

You can add elements to a list using:

- append() → adds an element at the end
- insert() → adds an element at a specific position
- extend() → adds multiple elements at once

Example:

```
fruits = ["apple", "banana"]

fruits.append("cherry")       # ['apple', 'banana', 'cherry']

fruits.insert(1, "orange")    # ['apple', 'orange', 'banana', 'cherry']
```

2. Deletion of Elements

You can remove elements using:

- remove() → deletes by value
- pop() → deletes by index
- del → deletes elements or the whole list

Example:

```
fruits = ["apple", "banana", "cherry"]

fruits.remove("banana")       # ['apple', 'cherry']
```

3. Updating Elements

You can change the value of a specific element using its index.

Example:

```
fruits = ["apple", "banana", "cherry"]

fruits[1] = "orange"

print(fruits)   # ['apple', 'orange', 'cherry']
```

4. Slicing

Slicing allows accessing a range of elements from a list.

Example:

```
numbers = [10, 20, 30, 40, 50]

print(numbers[1:4])   # Output: [20, 30, 40]
```

4. Tuple

1. Introduction to tuples, immutability

A tuple in Python is a collection data type that can store multiple items in a single variable, similar to a list. Tuples are created by placing elements inside parentheses ( ), separated by commas.

Example:

```
my_tuple = (10, 20, 30, "apple", True)
```

Tuples can contain elements of different data types such as integers, strings, or even other tuples.

Immutability of Tuples

The main feature that makes tuples different from lists is their immutability.

- Immutability means that once a tuple is created, its elements cannot be changed, added, or removed.
- This makes tuples faster and more secure than lists when you want data that should not be modified.

Example:

```
my_tuple = (1, 2, 3)

my_tuple[1] = 5
```

2 Creating and accessing elements in a tuple.

A tuple in Python is an ordered collection of elements enclosed in parentheses ( ) and separated by commas. Tuples can store values of different data types, such as numbers, strings, or even other tuples.

1. Creating a Tuple

You can create a tuple in different ways:

Example:

```
# Creating a tuple

numbers = (10, 20, 30, 40)

fruits = ("apple", "banana", "cherry")
```

mixed = (1, "hello", 3.5, True)

- You can also create a single-element tuple by adding a comma after the value:
- single = (5,)

2. Accessing Elements

Elements in a tuple can be accessed using index numbers.
Indexing in tuples starts from **0** for the first element.

Example:

fruits = ("apple", "banana", "cherry")

print(fruits[0]) # Output: apple

print(fruits[2]) # Output: cherry

You can also use negative indexing to access elements from the end:

print(fruits[-1]) # Output: cherry

3.Basic operations with tuples: concatenation, repetition, membership.

Tuples in Python support several basic operations that make them easy to use and combine. The most common operations are concatenation, repetition, and membership.

1. Concatenation (+ Operator)

- Concatenation means joining two or more tuples together to form a new tuple.
- The + operator is used for this purpose.

Example:

tuple1 = (1, 2, 3)

tuple2 = (4, 5)

result = tuple1 + tuple2

print(result)   # Output: (1, 2, 3, 4, 5)

2. Repetition (* Operator)

- Repetition is used to repeat the elements of a tuple multiple times.
- The * operator is used for this.

Example:

tuple1 = ("apple", "banana")

print(tuple1 * 2) # Output: ('apple', 'banana', 'apple', 'banana')

3. Membership (in / not in Operator)

- Membership operators are used to check if an element is present in a tuple.

Example:

fruits = ("apple", "banana", "cherry")

print("apple" in fruits)   # Output: True

print("grape" not in fruits) # Output: True

5. Accessing Tuples

1.Accessing tuple elements using positive and negative indexing

In Python, tuples are ordered collections, meaning each element has a specific position or index number. We can access elements of a tuple using indexing — either positive or negative.

1. Positive Indexing

- In positive indexing, counting starts from **0** for the first element.
- The index increases by 1 for each next element.
- It is used to access elements from the beginning of the tuple.

Example:

fruits = ("apple", "banana", "cherry", "mango")

print(fruits[0]) # Output: apple

print(fruits[2]) # Output: cherry

Here,

- fruits[0] → first element
- fruits[1] → second element

2. Negative Indexing

- In negative indexing, counting starts from -1 for the last element, **-**2 for the second last, and so on.
- It is used to access elements from the end of the tuple.

Example:

fruits = ("apple", "banana", "cherry", "mango")

print(fruits[-1]) # Output: mango

print(fruits[-3]) # Output: banana

Here,

- fruits[-1] → last element
- fruits[-2] → second last element

2. Slicing a tuple to access ranges of elements

Slicing in Python allows us to access a range of elements from a tuple instead of a single element. It helps in retrieving a portion (sub-tuple) from the original tuple without modifying it.

Syntax:

tuple_name[start:end]

- start → index where slicing begins (included)
- end → index where slicing ends (excluded)

Example:

numbers = (10, 20, 30, 40, 50, 60)

print(numbers[1:4])   # Output: (20, 30, 40)

Here, slicing starts from index 1 and ends before index 4.

Omitting Start or End Index:

You can skip the start or end index:

print(numbers[:3])   # Output: (10, 20, 30)   → from beginning to index 2

print(numbers[2:])   # Output: (30, 40, 50, 60) → from index 2 to end

Using Step Value:

You can also use a step to skip elements.

print(numbers[0:6:2]) # Output: (10, 30, 50)

6. Dictionaries

1.Introduction to dictionaries: key-value pairs.

A dictionary in Python is an unordered collection of data stored in the form of key–value pairs. Each item in a dictionary has a key and its corresponding value, separated by a colon : and enclosed within curly braces { }.

Syntax:

dictionary_name = {key1: value1, key2: value2, key3: value3}

Example:

student = {"name": "Rahul", "age": 20, "course": "Python"}

Here:

- "name", "age", and "course" are keys
- "Rahul", 20, and "Python" are their values

You can access values using their keys:

print(student["name"]) # Output: Rahul

Key Points:

- Keys must be unique and immutable (like strings or numbers).
- Values can be of any data type and can be duplicated.
- Dictionaries are useful for storing and retrieving data by names instead of positions.

2. Accessing, adding, updating, and deleting dictionary elements

A dictionary in Python stores data as key–value pairs, and it allows you to easily access, add, update, and delete elements using keys.

1. Accessing Elements

You can access a value in a dictionary using its key name inside square brackets [ ] or with the get() method.

Example:

student = {"name": "Rahul", "age": 20, "course": "Python"}

print(student["name"])   # Output: Rahul

print(student.get("course")) # Output: Python

2. Adding Elements

You can add a new key–value pair by assigning a value to a new key.

Example:

student["city"] = "Surat"

print(student)

# Output: {'name': 'Rahul', 'age': 20, 'course': 'Python', 'city': 'Surat'}

3. Updating Elements

You can update the value of an existing key directly or by using the update() method.

Example:

```
student["age"] = 21

student.update({"course": "Advanced Python"})

print(student)

# Output: {'name': 'Rahul', 'age': 21, 'course': 'Advanced Python'}
```

4. Deleting Elements

You can delete items from a dictionary using:

- del statement → removes a specific key
- pop() method → removes a key and returns its value
- clear() → removes all elements

Example:

```
del student["city"]

student.pop("age")

print(student)

# Output: {'name': 'Rahul', 'course': 'Advanced Python'}
```

3) Dictionary methods like keys(), values(), and items().

Python provides several built-in dictionary methods that help to access and view data easily. The most commonly used ones are keys(), values(), and items().

1. keys()

- The keys() method returns a list-like view of all the keys present in the dictionary.

Example:

```
student = {"name": "Rahul", "age": 20, "course": "Python"}

print(student.keys())

# Output: dict_keys(['name', 'age', 'course'])
```

2. values()

- The values() method returns a view object containing all the values in the dictionary. Example:

```
print(student.values())

# Output: dict_values(['Rahul', 20, 'Python'])
```

3. items()

- The items() method returns a view object that contains each key–value pair as a tuple.
  Example:

```
print(student.items())

# Output: dict_items([('name', 'Rahul'), ('age', 20), ('course', 'Python')])
```

7. Working with Dictionaries

1) Iterating over a dictionary using loops

In Python, iterating over a dictionary means going through its keys, values, or key–value pairs one by one using a for loop. This helps to access and process each element easily.

1. Iterating Over Keys

By default, a for loop iterates through the keys of a dictionary.
Example:

```
student = {"name": "Rahul", "age": 20, "course": "Python"}


for key in student:

  print(key)

# Output: name, age, course
```

2. Iterating Over Values

To loop through only the values, use the values() method.
Example:

```
for value in student.values():

  print(value)

# Output: Rahul, 20, Python
```

3. Iterating Over Key–Value Pairs

To access both the key and value at the same time, use the items() method.
Example:

```
for key, value in student.items():

    print(key, ":", value)

# Output:

# name : Rahul

# age : 20

# course : Python
```

2.Merging two lists into a dictionary using loops or zip()

In Python, you can combine (merge) two lists — one containing keys and the other containing values — to form a dictionary. This can be done using a loop or the zip() function.

1. Using a Loop

You can use a for loop with the range() function to pair elements from both lists and add them to a dictionary.
Example:

```
keys = ["name", "age", "city"]

values = ["Rahul", 20, "Surat"]


my_dict = {}

for i in range(len(keys)):

    my_dict[keys[i]] = values[i]


print(my_dict)

# Output: {'name': 'Rahul', 'age': 20, 'city': 'Surat'}
```

2. Using the zip() Function

The zip() function combines two lists element-wise, creating pairs that can easily be converted into a dictionary using the dict() constructor.
Example:

```
keys = ["name", "age", "city"]

values = ["Rahul", 20, "Surat"]
```

```
my_dict = dict(zip(keys, values))

print(my_dict)

# Output: {'name': 'Rahul', 'age': 20, 'city': 'Surat'}
```

3. Counting occurrences of characters in a string using dictionaries.

In Python, we can use a dictionary to count how many times each character appears in a string.
Each character becomes a key, and its count (frequency) becomes the value.

1. Using a Loop

We can iterate through each character in the string and update the dictionary count.

Example:

```
text = "hello"

count = {}


for char in text:

    if char in count:

        count[char] += 1   # Increase count if character exists

    else:

        count[char] = 1    # Add new character with count 1


print(count)

# Output: {'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

2. Explanation:

- A dictionary named count is created to store characters and their frequencies.
- The loop checks if the character already exists in the dictionary:
    - If yes, its value is incremented by 1.
    - If no, it is added to the dictionary with value 1.

8. Functions

1. Defining functions in Python

A function in Python is a block of reusable code that performs a specific task.
Functions help make programs modular, organized, and easy to maintain.

In Python, a function is defined using the def keyword, followed by the function name and parentheses ( ).

Syntax:

```
def function_name(parameters):
    # function body
    statement(s)
    return value
```

Example:

```
def greet():
    print("Hello, welcome to Python!")


greet()
# Output: Hello, welcome to Python!
```

Explanation:

- def → used to define the function.
- function_name → the name used to call the function later.
- parameters → optional inputs to the function.
- return → sends a value back to the caller (optional).

2. Different types of functions: with/without parameters, with/without return values.

In Python, functions are categorized based on whether they take parameters (inputs) and return values (outputs).
There are four main types of functions:

1. Function Without Parameters and Without Return Value

- These functions neither take any input nor return any output.
- They only perform a task when called.

Example:

```
def greet():
    print("Hello, welcome!")
greet()
```

Output:

```
Hello, welcome!
```

2. Function With Parameters and Without Return Value

- These functions take input values (parameters) but don’t return anything.
- They perform operations and display the result directly.

Example:

```
def add(a, b):
    print("Sum:", a + b)
add(5, 3)
```

Output:

```
Sum: 8
```

3. Function Without Parameters and With Return Value

- These functions do not take inputs but return a result.

Example:

```
def get_name():
    return "Rahul"
print(get_name())
```

Output:

```
Rahul
```

4. Function With Parameters and With Return Value

- These functions take inputs and also return a result.
- This is the most flexible and commonly used type.

Example:

```
def multiply(a, b):
    return a * b
result = multiply(4, 5)
```

```
print(result)
```

Output:

```
20
```

3. Anonymous functions (lambda functions).

An anonymous function in Python is a function without a name, also called a lambda function.
It is used to write small, one-line functions for short and simple operations.

Syntax:

```
lambda arguments: expression
```

- The lambda keyword is used to create the function.
- It can take any number of arguments, but only one expression.
- The expression's result is automatically returned.

Example 1:

```
square = lambda x: x * x

print(square(5))
```

Output:

```
25
```

Example 2:

```
add = lambda a, b: a + b

print(add(3, 7))
```

Output:

```
10
```

Use of Lambda Functions:

- Used in short calculations.
- Often used with functions like map(), filter(), and reduce().

Example:

```
numbers = [1, 2, 3, 4]

doubled = list(map(lambda x: x * 2, numbers))

print(doubled)
```

Output:

```
[2, 4, 6, 8]
```

9. Modules

1) Introduction to Python modules and importing modules

A module in Python is a file that contains Python code, such as functions, variables, or classes, that can be reused in other programs.
Modules help in organizing large programs into smaller, manageable, and reusable parts.

For example, if you have a set of functions used often, you can put them in a separate file (module) and use them whenever needed.

1. Creating a Module

A module is simply a Python file (.py).
Example:

```
# mymodule.py

def greet(name):

    return f"Hello, {name}!"
```

2. Importing a Module

To use a module in another program, you use the import statement.

Example:

```
import mymodule

print(mymodule.greet("Rahul"))
```

Output:

```
Hello, Rahul!
```

3. Importing Specific Items

You can import only specific functions or variables from a module using the from keyword.

Example:

```
from math import sqrt

print(sqrt(16))
```

Output:
4.0

4. Built-in Modules

Python provides many built-in modules like:

- math – for mathematical operations
- random – for random numbers
- datetime – for date and time
- os – for operating system operations

Example:

```
import math

print(math.pi)
```

Output:
```
3.141592653589793
```

2. Standard library modules: math, random.

Python comes with a rich standard library that includes many useful modules.
Two commonly used modules are math and random, which provide functions for mathematical and random number operations.

1. math Module

The math module provides mathematical functions like square root, power, trigonometry, and constants such as π (pi) and e.

Example**:**

```
import math


print(math.sqrt(25))    # Square root → 5.0

print(math.pow(2, 3))   # Power → 8.0

print(math.pi)          # Value of π → 3.141592653589793
```

Common math module functions:

Function	Description	Example
sqrt(x)	Returns square root of x	math.sqrt(16) → 4.0
pow(x, y)	x raised to the power y	math.pow(2, 3) → 8.0
ceil(x)	Rounds number up	math.ceil(2.3) → 3
floor(x)	Rounds number down	math.floor(2.8) → 2

2. random Module

The random module is used to generate random numbers or select random items from a list. It is useful in games, simulations, and testing.

Example:

```
import random


print(random.randint(1, 10))     # Random integer between 1 and 10
print(random.random())         # Random float between 0 and 1
print(random.choice(['red', 'blue', 'green']))  # Randomly select item
```

Common random module functions:

Function	**Description**	**Example**
random()	Returns random float between 0 and 1	random.random()
randint(a, b)	Returns random integer between a and b	random.randint(1, 100)
choice(seq)	Returns random element from a sequence	random.choice(list)

3) Creating custom modules

In Python, a custom module is a user-defined file that contains Python code such as functions, variables, or classes which can be reused in other programs.
Creating custom modules helps keep code organized, modular, and reusable.

1. Creating a Module

A module is simply a Python file (.py) containing some definitions.
Example:

```
# mymodule.py

def greet(name):

    return f"Hello, {name}!"


def add(a, b):

    return a + b
```

Here, mymodule.py is our **custom module**.

2. Importing a Custom Module

To use the functions or variables from your module, you can import it into another Python file using the import statement.

Example:

```
# main.py

import mymodule


print(mymodule.greet("Rahul"))

print(mymodule.add(5, 10))
```

Output:

```
Hello, Rahul!

15
```

3. Importing Specific Functions

You can import only certain functions using the **from** keyword.

Example:

```
from mymodule import greet

print(greet("Bhautik"))
```

4. Advantages of Custom Modules

- Code reusability (use the same functions in many programs)
- Better organization of large projects
- Easy to maintain and debug